Department Of Computer Science and Engineering

Course Title: Operating System Lab

Course Code: CSE 406

Title: Priority Scheduling

Submitted To:

Atia Rahman Orthi
Lecturer
Department Of Computer Science & Engineering

Submitted By:

Jannatun Saumoon
ID: 21201066
Sec: B2

1)Problem Statement:

Input:

Process ID	Priority	Arrival Time	Burst Time
1	10	0	5
2	20	1	4
3	30	3	2
4	40	4	1

Output:

Process ID	Priority	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
1	10	0	5	12	12	7
2	20	1	4	9	8	4
3	30	3	2	7	4	2
4	40	4	1	5	1	0

Execution Order: P1 → P2 → P4 → P3 → P2 → P1 → P1

2)Steps:

1. Input process list: PID, Priority, Arrival Time, Burst Time
2. Sort by arrival time and priority
3. Initialize time, remaining burst times, and result trackers
4. While not all complete:
    - Pick ready processes (arrived, not done)
    - Select the one with highest priority
    - Run it for time_quantum or less
    - Update time and remaining burst
    - If finished, calculate CT, TAT, WT

5.After loop ends, print:

- Process table (PID, Priority, AT, BT, CT, TAT, WT)
- Execution order

3)Source Code:

```
def priority_scheduling(processes, time_quantum):
    time, completed = 0, 0
    n = len(processes)
    remaining_bt = [p[3] for p in processes]
    processes.sort(key=lambda x: (x[2], -x[1]))

    executed_order = []
    process_info = []

    while completed < n:

        ready_queue = [p for p in processes if p[2] <= time and remaining_bt[processes.index(p)] > 0]

        if ready_queue:

            ready_queue.sort(key=lambda x: -x[1])


            process = ready_queue[0]
            pid_index = processes.index(process)
            executed_order.append(process[0])


            exec_time = min(time_quantum, remaining_bt[pid_index])
```

```
            exec_time = min(time_quantum, remaining_bt[pid_index])


            remaining_bt[pid_index] -= exec_time
            time += exec_time


            if remaining_bt[pid_index] == 0:
                completed += 1
                completion_time = time
                turnaround_time = completion_time - process[2]  # CT - AT
                waiting_time = turnaround_time - process[3]  # TAT - BT

                process_info.append([process[0], process[1], process[2], process[3], completion_time, turnaround_time, waiting_time])
        else:
            time += 1
```

```
        else:
            time += 1

    # Output the results
    print("\nPID\tPR\tAT\tBT\tCT\tTAT\tWT")
    for p in sorted(process_info, key=lambda x: x[0]):  # Sort by PID
        print(f"{p[0]}\t{p[1]}\t{p[2]}\t{p[3]}\t{p[4]}\t{p[5]}\t{p[6]}")

    # Print execution order
    print("\nExecution Order:", " → ".join(f"P{p}" for p in executed_order))

# Input Data (PID, Priority, Arrival Time, Burst Time)
process_list = [
    [1, 10, 0, 5],
    [2, 20, 1, 4],
    [3, 30, 3, 2],
    [4, 40, 4, 1]
]

time_quantum = 2
priority_scheduling(process_list, time_quantum)
```

Output:

```
PID     PR      AT      BT      CT      TAT     WT
1       10      0       5       12      12      7
2       20      1       4       9       8       4
3       30      3       2       7       4       2
4       40      4       1       5       1       0

Execution Order: P1 → P2 → P4 → P3 → P2 → P1 → P1
```

4)Conclusion:

The code accurately simulates a hybrid scheduling strategy that balances fair CPU usage (via time quantum) with priority-based execution. It effectively tracks and outputs all key metrics and the execution order, providing a clear view of how CPU time is distributed. For standard behavior (where lower priority number = higher priority), a small adjustment is needed in the sorting logic.